Prefetching

An introduction to and analysis of Hardware and Software based Prefetching

Jun Yi Lei
Robert Michael Allen Jr.
Outline

- What is Prefetching
- Background
- Software vs Hardware
- Hardware Implementation
- Current and Future Implications
- Summary
- Questions
What is Prefetching?

- Process of bringing data into a higher level of memory before it is actually needed for processing
- Algorithms are designed to predict future data requests
- Reduces miss rate

AMD's Bulldozer, an x86 core processor
Background
(History)

- Intel's i486 in 1989 was the first chip to have an internal L1 cache.
- Prefetching however was not utilized until some years later (in the traditional sense with caches)
- Prefetching uses unused memory bandwidth to perform operations.
Background
(Requirements to Prefetch)

- Present working set must not be replaced by future set
- Must disjoint from the data cache ... avoids increasing bandwidth and complexity of the cache
- Must be expandable
- Must keep the data until it is unnecessary
Background
(When to Prefetch)

- Data should arrive to its destination before its needed
- Prefetch on miss?
- Prefetch at regular intervals?
- Prefetch dynamically based on program history?
Background
(How to Prefetch)
Background
(Who Wants to Prefetch)

- Push-based prefetching (simultaneous thread requests for prefetching)
- Processor Side
- Memor Side
Software vs Hardware
(Software Definition)

- Software Prefetching
  - Prefetching techniques performed by the compiler or by the programmer
  - Usually can prefetch instructions
  - Utilizes prefetch input queue (PIQ) in certain architectures
  - Compiler assisted prefetching in loops – Stanford University Intermediate Form (SUIF)
Software vs Hardware
(Hardware Definition)

- Hardware Prefetching
  - Hardware detect access patterns and decide when to perform prefetching
  - Usually can prefetch future data
  - Prefetch into the prefetch buffer or free cache
Hardware Implementation (Overview)

- Two of many algorithms
  - Sequential Prefetching
  - Dependence Based Prefetching for Linked Data Structures (LDS)
Hardware Implementation (Sequential Prefetching)

- Utilizes principle of spatial locality to prefetch data
- Fetches number of cache blocks each time cache is accessed
- Benefits from spatial locality w/o problems associated with large cache blocks
- Simplest scheme is the One Block Look-ahead (OBL) approach
  - Fetch block n+1 when block n is accessed
Hardware Implementation (Sequential Prefetching)  
OBL – Prefetch on Miss

- Fetches block \( n + 1 \) when block \( n \) is requested from memory
- Requires a cache miss to prefetch
Hardware Implementation
(Sequential Prefetching)
OBL – Tagged Prefetch

- Utilizes a tag bit to detect when a block is demand fetched or prefetched.
- Does not require cache miss to trigger prefetch
- Allows for history based performance evaluation
  - Can keep track of number of block demand fetches vs block prefetches
Hardware Implementation (Sequential Prefetching)
OBL - Sequential Dynamic Dynamic Prefetch

- Instead of fetching block n+1, fetch block n+1 to n+K
- K is highly dependent upon the degree of spatial locality
  - Small K's – insufficient performance improvement
  - Large K's – cache pollution and collision
When considering the access pattern shown in earlier slides...

- Prefetch-on-Miss: Results in a cache miss for every other block... only improves performance by 50% at most
- Tagged Prefetch or Sequential Prefetch with K: Results in only one cache miss
Growing number of important, nonnumeric applications employ LDSs

Databases commonly use index trees and hash tables

Programs also allow for dynamic allocation and de-allocation

LDSs can be linear, binary trees, multi-way...
Hardware Implementation (Dependence Based Prefetching for LDS)

Concept

- Based on finding pointer loads and establishing dependence relationships
- “Producer Loads” vs “Consumer Loads”
- First load (producer) gets the address which the second load (consumer) uses to retrieve the actual value
- Prefetching would occur on the first load and get the address and the value that the second load will need.
Hardware Implementation
(Dependence Based Prefetching for LDS)
Main Components

- Correlation Table (CT)
  - Stores Dependence information about loads
- Potential Producer Window (PPW)
  - Maintains the list of most recently loaded values and the corresponding load instruction
- Prefetch Request Queue (PQR)
  - Buffers requests for prefetch
- Prefetch Buffer (PB)
  - Temporarily holds prefetch blocks
Hardware Implementation (Dependence Based Prefetching for LDS) Prefetching Process

- Requests enqueued in the PRQ are dequeued when they can be serviced by the memory system.

- The PB (not shown) attempts to extract the block from L1 cache, issuing a request to L2 cache on a miss.
Hardware Implementation (Dependence Based Prefetching for LDS) Dependency Information

- Dependency information is stored in the CT
  - prod.add, con.add, tmpl
  - prod.add = producing load instruction's PC address
  - con.add = consuming load instruction's PC address
  - tmpl = condensed form of the consuming load
- Need to create producer-consumer relationship... maintain a list of loaded values and their corresponding load instructions... stored in the PPW
  - addrval, prod.add
  - addrval = the base address used by the consumer
Hardware Implementation (Dependence Based Prefetching for LDS) Creating Correlations in CT

- Correlations are created at instruction commit time
- As a load commits, check to see if there's a potential producer in the PPW
- If a match exists, an entry in the CT is created
- The value loaded by the current load is entered along with the current PC into the PPW as a new addrval-prod.add pair
Hardware Implementation
(Dependence Based Prefetching for LDS)
Creating Correlations in CT

Piece of code that is responsible for traversing a LDS

a) Source code (C code)
b) Machine code equivalent
c) List layout in memory
Hardware Implementation
(Dependence Based Prefetching for LDS)
Creating Correlations in CT

<table>
<thead>
<tr>
<th>Line 1:</th>
<th>PPW</th>
<th>CT</th>
<th>Memory</th>
<th>Prefetching</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*(F+5)=A2</td>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>*(A2+2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line 3:</td>
<td>$24</td>
<td>*(A1+8)=Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*(A2+2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*(F+4)=A1</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line 6:</td>
<td>$24</td>
<td>*(A1+8)=Y</td>
<td></td>
<td>*(X+4) or *(X+8) is prefetched</td>
</tr>
<tr>
<td></td>
<td>*(A1+4)=X</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0*(S15)=Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*(F=3)=A2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line 3:</td>
<td>$15</td>
<td>*(X+8)=V</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*(A1+4)=X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*(X+8)=V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line 4 :</td>
<td></td>
<td>*(V)=V1</td>
<td></td>
<td>*(X1+4) or *(X1+8) is prefetched</td>
</tr>
<tr>
<td></td>
<td>*(V)=V1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*(A1+4)=X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line 6 :</td>
<td>$24</td>
<td>*(X+8)=V</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*(V)=V1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*(X+4)=X1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hardware Implementation
(Dependence Based Prefetching for LDS)
Creating Correlations in CT

- The average speedup for a system based on Dependence Based Prefetching for a 1KB PB is 10%
- This significantly outperforms a basic system with an extra 32KB of data cache. Increasing the PB past a certain point (~32KB) shows diminishing returns.
- CT however is implemented as a multi-way set associative cache and not all pointer loads are actually address loads.
- Unnecessary prefetches maybe initiated in this scheme.
Several types of prefetchers are included in Core:

The small suns represent the 8 prefetchers of the Core 2 Duo.
Several types of prefetchers are included in Core:

- the instruction prefetcher pre loads instructions in the instruction L1 cache based on branching prediction results. Each of the two cores has one.

- the IP prefetcher scrutinizes historical reading in order to have an overall diagram and loads "foreseeable" data in L1 cache. Each core also has one.

- The DCU prefetcher detects multiple reading from a single cache line for a determined period of time and decides to load the following line in the L1 cache. One per core as well.

- the DPL prefetcher has a similar functioning to the DCU. The only difference is that it detects requests on two successive cache lines (N and N+1) and is triggered if the reading of the line N+2 moves from the central memory to cache L2. The cache L2 has two of them, which are shared dynamically between the two cores.
Performance Evaluation on a baseline system

<table>
<thead>
<tr>
<th>Evaluation Metric</th>
<th>System Parameters</th>
<th>Default Value</th>
<th>System Parameters</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache tag array</td>
<td>double ported</td>
<td></td>
<td>Bus transaction</td>
<td>non-split</td>
</tr>
<tr>
<td>Cache data array</td>
<td>single ported</td>
<td></td>
<td>Bus width</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Cache size</td>
<td>64K bytes</td>
<td></td>
<td>Memory latency</td>
<td>16 CPU cycles</td>
</tr>
<tr>
<td>Cache block size</td>
<td>64 bytes</td>
<td></td>
<td>Number of memory banks</td>
<td>4</td>
</tr>
<tr>
<td>Cache associativity</td>
<td>8</td>
<td></td>
<td>Memory bank width</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Cache type</td>
<td>split</td>
<td></td>
<td>Prefetch lookahead distance</td>
<td>1 cache block</td>
</tr>
</tbody>
</table>
## Performance Evaluation Cont.

<table>
<thead>
<tr>
<th>CPU Stall Reasons</th>
<th>Cache Prefetching Strategies</th>
<th>Prefetch Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>none</td>
<td>always</td>
</tr>
<tr>
<td>cache misses</td>
<td>99.27</td>
<td>51.56</td>
</tr>
<tr>
<td>prefetch (cache)</td>
<td>0.00</td>
<td>33.97</td>
</tr>
<tr>
<td>prefetch (bus)</td>
<td>0.00</td>
<td>13.94</td>
</tr>
<tr>
<td>prefetch (mem)</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>write buffer</td>
<td>0.72</td>
<td>0.52</td>
</tr>
</tbody>
</table>
Citations


Summary

- What is Prefetching
- Background
- Software vs Hardware
- Hardware Implementation
- Current and Future Implications
- Summary
- Questions
Questions